Some changes in `snow` and R

Luke Tierney

Department of Statistics & Actuarial Science
University of Iowa

December 13, 2007

Some Comments on the Course

- Some new ideas for me:
  - Grid computing
  - MapReduce
  - OpenMP
- Some opportunities:
  - Rethink some aspects of `snow` design
  - Make progress on parallel vectorized arithmetic for R

Some Open snow Issues

- better error handling
- integrating load balancing into all functions
- R-level collection of timing information
- non-parallel testing framework
- persistent data on nodes
- limited inter-node communication
- sensible handling of user interrupts

- Errors in `clusterXYZ` functions used to be returned as `try-error` objects.
- New version will signal an error on the master if there is an error on any node.
- This is better but not always ideal: Sometimes one good result is enough.

Some snow Changes
Argument Length Limits

- Originally `clusterApply` required at most as many elements as cluster nodes.
- Longer lists can be handled by `clusterApplyLB`
- This does not provide a deterministic option for longer vectors.
- It also leaves out the new `clusterMap` function.
- The new version allows longer vectors.
- By default nodes are recycled.
- This produces deterministic job/node assignments.

Some snow Changes

Integrating Load Balancing

- Load balancing might be useful with functions other than just `clusterApply`.
- Motivated by OpenMP, one option is to
  - allow a `SCHEDULE` argument with values `"static"` or `"dynamic"` got all functions
  - a variant is to allow a boolean `LoadBalance` argument
  - have the `parXYZ` functions take a `ChunkSize` argument
- Default will be static.
- Open question: can parallel RNG streams be tied to jobs in a simple way so results from with load-balancing can be reproducible?

Some snow Changes

Collecting Timing Information in R

- xpvm is very useful, but requires pvm.
- xmpi is similar but restricted to LAM.
- An alternative is to collect timing information in R:
  - In the master record the start and finish of each send/recv.
  - In the nodes, record duration of computation and send back with result.
- Need to decide interface for collecting the data.
- One possibility, motivated by Rprof:
  - `traceCluster(file=''foo.trace'')` to start recording to a file
  - `traceCluster(NULL)` to stop recording
- Then need some functions to read trace file and produce graphs.
- Will experiment with this in the next month or so.

- It may be useful to have a “null cluster” so that

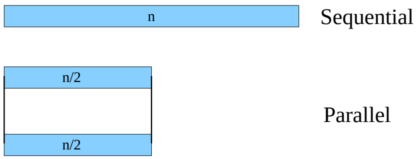
```
cl <- makeNULLcluster(4)
clusterApply(cl, ...)
```

  works within the master process
- This will help with
  - debugging
  - running small jobs without parallel complications
- Some detail issues:
  - should the cluster size argument matter?
  - should random number streams behave as in the parallel version?

- It can be useful to leave large computed values on nodes for further computation.
- Global variables can be used but are awkward and not very clean.
- A better option may be to have a means of returning only a remote object reference.
- These remote objects can then be passed to subsequent calls.
- Once the master no longer has a reference to a remote object it can be garbage collected.

- A next step is to allow remote data to move between nodes, e.g.
  - think of the nodes as arranged in a circle
  - each node passes its data to the node to its left
- This leads to a model called Bulk Synchronous Parallel (BSP) computing.
- BSP has some interesting theoretical properties
  - a cost model for comparing parallel algorithms in terms of simple machine parameters
  - deadlock-free
- BSP has been used as the basis for parallel computing support for several high level languages.
- An initial (and maybe inefficient) BSP extension may be available soon.

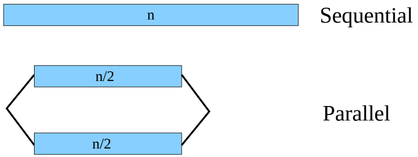
- Basic idea for computing `f(x[1:n])` on a two-processor system:
  - Run two worker threads.
  - Place half the computation on each thread.
- Ideally this would produce a two-fold speed up.



Parallelizing Vector Operations

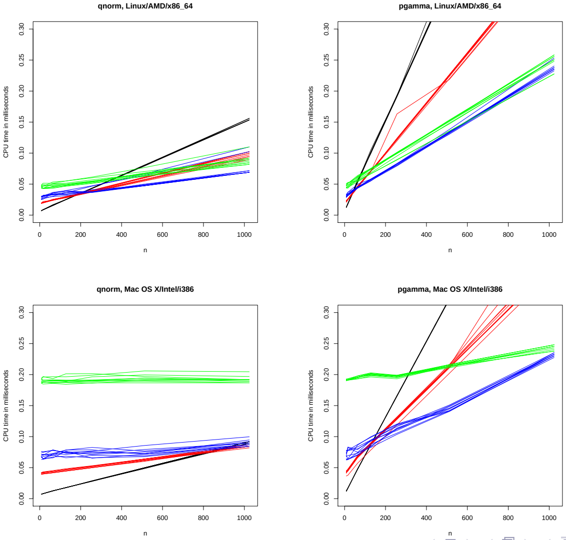
A More Realistic View

- Reality is a bit different:



- There is synchronization overhead.
- Use of shared resources is sequential (memory, bus, . . .)
- Parallelizing will only pay off if n is large enough.
  - For some functions, e.g. `qbeta`, $n \approx 10$ may be large enough.
  - For some, e.g. `qnorm`, $n \approx 1000$ is needed.
  - For basic arithmetic operations $n \approx 30000$ may be needed.
- Careful tuning to insure improvement will be needed.
- Some aspects will depend on architecture and OS.

Parallelizing Vector Operations
Some Experimental Results



Some observations:

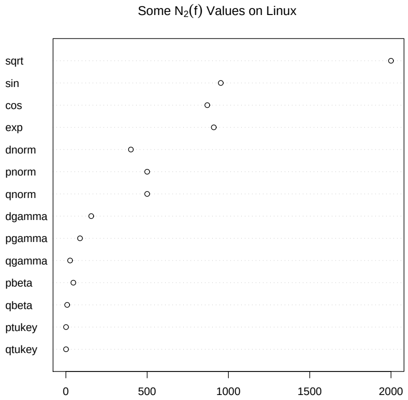
- Times are roughly linear in vector length.
- Intercepts on a given platform are roughly the same for all functions.
- If the slope for P processors is s_P, then at least for $P = 2$ and $P = 4$,

$$s_P \approx s_1/P$$

- Relative slopes of functions seem roughly independent of OS/architecture.

- A simple strategy:
  - Compute relative slopes once, or average across several setups.
  - Base line is a single element `dnorm` computation.
  - For each OS/architecture combination compute the intercepts.
  - Estimate the values $N_2(f)$ such that using $P = 2$ is faster if $n > N_2(f)$.
  - Use $N_4(f) = 2N_2(f)$ and $N_8(f) = 4N_2(f)$.
- Some intercepts, in units of a single element `dnorm` computation:
  - about 200 for Linux/AMD/x86_64
  - about 500 for Mac OS X 10.4/Intel/i386
  - between 300 and 400 for Win32/Intel(?)

Parallelizing Vector Operations



- Need to use threads
- One possibility: using raw `pthreads`
- Better choice: use Open MP
- Open MP consists of
  - compiler directives (`#pragma` statements in C)
  - a runtime support library
- Most commercial compilers support Open MP.
- gcc 4.2 supports Open MP.
- Redhat has back-ported Open MP into gcc 2.4.1 on RH, Fedora.
- MinGW also supports Open MP; an additional `pthreads` download is needed.

Parallelizing Vector Operations
Implementation

- Basic loop for a one-argument function:

```
#pragma omp parallel for if (P > 1) num_threads(P) \
        default(shared) private(i) reduction(&&:naflag)
    for (i = 0; i < n; i++) {
        double ai = a[i];
        MATH1_LOOP_BODY(y[i], f(ai), ai, naflag);
    }
```

- Steps in converting to Open MP:
  - check `f` is thread-safe; modify if not
  - rewrite loop to work with the Open MP directive
  - test without Open MP, then enable Open MP

- Some things that are not thread-safe:
  - use of global variables
  - R memory allocation
  - signaling warnings and errors
  - user interrupt checking
  - creating internationalized messages (calls to `gettext`)
- Random number generation is also problematic.
- Functions in `nmath` that have not been parallelized yet:
  - Bessel functions
  - Wilcoxon, signed rank functions
  - random number generators

Parallelizing Vector Operations
Availability

- Package pnmath is available at

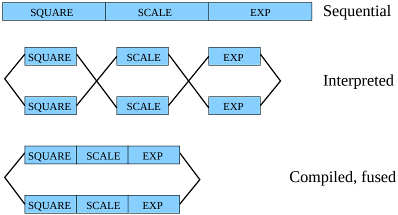
  `http://www.stat.uiowa.edu/~luke/R/experimental/`
- This requires a version of gcc that
  - supports Open MP
  - allows `dlopen` to be used on `libgomp.so`

  Our current systems don't satisfy this.
- A version using just pthreads is available in pnmath0. This should work on current R on our systems.
- Loading these packages replaces builtin operations by parallelized ones.
- For Linux, Mac OS X predetermined intercept calibrations are used.
- For other platforms a calibration test is run at package load time.
- The calibration can be run manually by calling `calibratePnmath`
- Hopefully we will be able to include this in R 2.7 or 2.8.

Connection to Compilation

- Developing a byte code compiler for R is an ongoing project.
- The `codetools` package is a by-product.
- Compilation will also be useful for parallelizing vector operations:
  - Many vector operations occur in compound expressions, like

    `exp(-0.5*x^2)`
  - A compiler may be able to fuse these operations:



- Compilation may also allow many simple uses of `apply` functions and `sweep` to be parallelized.

Challenges

- Tuning issues:
  - Hardware/OS plays a role.
  - Competing system usage may be important.
  - Performance may vary with inputs.
  - Load balancing may be useful.
- Error handling and user interrupts.
- Parallelization interface for package use.
- Extensible byte code for package use.
- Generic functions and non-default methods.
- Declarations may be useful.